Extending R with classes, modules, procs, lambdas, oh my!

Rodrigo Botafogo Daniel Mossé - University of Pittsburgh

November 19th, 2018 (narrative updated for Galaaz 2.0, 2026)

Contents

1 Introduction

This paper introduces and compares Galaaz with R's S4. It is a shameless rip off of "A '(not so)' Short Introduction to S4" by Christophe Genolini and follows the same structure and examples presented there.

Galaaz is a Ruby Gem (library) that allows very tight integration between Ruby and R.
Its integration is tighter and more transparent than what one can get between RinRuby or similar solutions in Python, such as PypeR, rpy2 and other similar solutions.

Galaaz 2.0 runs on **JRuby** and drives **GNU R** through a **bridge**, so Ruby code can create and manipulate R objects and call R functions while staying idiomatic Ruby. An earlier prototype used Oracle's **GraalVM** with **TruffleRuby** and **FastR**; that stack is historical and is **not** what current Galaaz targets.

2 Bases of Object Programming

In this paper, we will start our discussion from Part II of "The (not so) Short Introduction to S4", which from now on we will reference as SS4 for "short S4". Interested readers are directed to this paper to understand the motivation and examples in that paper. In this paper we will present the S4 code from SS4 and then the same code in Ruby/Galaaz. We will not comment on the S4 code, as all the comments can be found in SS4, we will only focus on the Ruby/Galaaz description.

S4 defines classes by using the setClass function:

3 Classes Declaration

```
# > setClass(
# + Class="Trajectories",
# + representation=representation(
# + times = "numeric",
# + traj = "matrix"
# + )
# + )
```

4 Instance Variables

In Ruby a class is defined by the keyword 'class'. Every class should start with a capital letter. S4 'slots' are called 'instance variables' in Ruby. Differently from R's S4, instance variables in Ruby do not have type information. It should be clear though, that S4 type information is also not a "compile" time type, since R is not compiled. The type is checked at runtime. The same checking can be done in Ruby and we will do it later in this document.

In the example bellow, we create class Trajectories with two instance variables, 'times' and 'matrix'. We will not go over the details of instance variables in Ruby, but here we created those variables with the keyword 'attr_reader' and a colom before the variables name:

```ruby
class Trajectories
  attr_reader :times
  attr_reader :matrix
end
```

In order to create a new instance of object Trajectories we call method new on the class and we can store the result in a varible (not an instance variable) as bellow:

```ruby
@traj = Trajectories.new
```

We now have in variable '@traj' a Trajectories object. In Ruby, printing variable 'traj' will only print the class name of the object and not it contents as in R.

```ruby
puts @traj
```

```
## #<RC::Trajectories:0x378c2595>
```

To see the contents of an object, one needs to access its components using the '.' operator:

```
puts @traj.times
```

5 Constructor

Since there is no content stored in 'times' nor 'matrix', nil is returned. In order to add a value in the variables, we need to add a constructor to the class Trajectories. In R, a constructor is build by default, in Ruby, this has to be created by adding a method called 'initialize'. In the example bellow, we will create the initializer that accepts two values, a 'times' value and a 'matrix' value and they are used to initialize the value of the instance variables:

```
class Trajectories

  attr_reader :times
  attr_reader :matrix

  #------------------------------------------------------------
  # Initializes the Trajectories class.  Takes two parameters
  # @param times
  # @param matrix
  #------------------------------------------------------------

  def initialize(times: nil, matrix: nil)
    @times = times
    @matrix = matrix
  end

end
```

Up to this point, everything described is pure Ruby code and has absolutely no relationship with R. We now want to create a Trajectories with a 'times' vector. Ruby has a vector class and we could use this class to create a vector and add it to the 'times' instance variable; however, in order to make use of R's functions, we want to create a R vector to add to 'times'. In Galaaz, creating R objects is done using the corresponding R functions by just preceding them with 'R.', i.e., R functions are all defined in Galaaz in the R namespace.

Since Galaaz is Ruby and not R, some syntax adjustments are sometimes necessary. For instance, in R, a range is represented as '(1:4)', in Ruby, the same range is represented as '(1..4)'. When passing arguments to an R function in R one uses the '=' sign after the slot name; in R, one uses the ':' operator after parameter's name as we can see bellow:

```
# Create a Trajectories passing a times vector, but no matrix parameter
@traj = Trajectories.new(times: R.c(1, 2, 3, 4))

# Create a Trajectories with times and matrix
@traj2 = Trajectories.new(times: R.c(1, 3), matrix: R.matrix((1..4), ncol: 2))
```

6 Access to Instance Variables (to reach a slot)

In order to access data in an instance variable the operator '.' is used. In R, a similar result is obtained by use of the '@' operator, but SS4 does not recommend its use. In Galaaz, the '.' operator is the recommended way of accessing an instance variable.

Now that we have created two trajectories, let's try to print its instance variables to see that everything is fine:

```
puts @traj.times
```

```
## [1] 1 2 3 4
```

We now have the expected value. Note that the 'times' vector is printed exactly as it would if we were using GNU R. Let's now take a look at variable 'traj2':

```
puts @traj2.times
puts
puts @traj2.matrix
```

```
## [1] 1 3
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

Let's now build the same examples as in SS4: Three hospitals take part in a study. The Pitié Salpêtriere (which has not yet returned its data file, shame on them!), Cochin and Saint-Anne. We first show the code in R and the corresponding Galaaz:

```
> trajPitie <- new(Class="Trajectories")
> trajCochin <- new(
+     Class= "Trajectories",
+     times=c(1,3,4,5),
+     traj=rbind (
+         c(15,15.1, 15.2, 15.2),
+         c(16,15.9, 16,16.4),
+         c(15.2, NA, 15.3, 15.3),
+         c(15.7, 15.6, 15.8, 16)
+     )
+ )
> trajStAnne <- new(
+     Class= "Trajectories",
+     times=c(1: 10, (6: 16) *2),
+     traj=rbind(
+         matrix (seq (16,19, length=21), ncol=21, nrow=50, byrow=TRUE),
+         matrix (seq (15.8, 18, length=21), ncol=21, nrow=30, byrow=TRUE)
+     )+rnorm (21*80,0,0.2)
+ )
```

This same code in Galaaz becomes:

```
@trajPitie = Trajectories.new

@trajCochin = Trajectories.new(times: R.c(1,3,4,5),
                               matrix: R.rbind(
                               R.c(15,15.1, 15.2, 15.2),
                               R.c(16,15.9, 16,16.4),
                               R.c(15.2, R::NA, 15.3, 15.3),
                               R.c(15.7, 15.6, 15.8, 16)))

@trajStAnne =
  Trajectories.new(times: R.c((1..10), R.c(6..16) * 2),
                   matrix: (R.rbind(
                            R.matrix(R.seq(16, 19, length: 21), ncol: 21,
                                     nrow: 50, byrow: true),
                            R.matrix(R.seq(15.8, 18, length: 21), ncol: 21,
                                     nrow: 30, byrow: true)) + R.rnorm(21*80, 0, 0.2)))
```

Let's check that the 'times' and 'matrix' instance variables were correctly set:

```
puts @trajCochin.times
puts
puts @trajCochin.matrix
puts
puts @trajStAnne.times
```

```
## [1] 1 3 4 5
##
##          [,1] [,2] [,3] [,4]
## g2_v2114 15.0 15.1 15.2 15.2
## g2_v2115 16.0 15.9 16.0 16.4
## g2_v2116 15.2   NA 15.3 15.3
## g2_v2117 15.7 15.6 15.8 16.0
##
##  [1]  1  2  3  4  5  6  7  8  9 10 12 14 16 18 20 22 24 26 28 30 32
```

We will not at this time print trajStAnne.matrix, since this is a huge matrix and the result would just take too much space. Later we will print just a partial view of the matrix.

7 Default Values

Default values are very useful and quite often used in Ruby programs. Although SS4 does not recommend its use, there are many cases in which default values are useful and make code simpler. We have already seen default values in this document, with the default being 'nil'. This was necessary in order to be able to create our constructor and passing it the proper values.

In the example bellow, a class TrajectoriesBis is created with default value 1 for times and a matrix with no elements in matrix.

```
class TrajectoriesBis

  attr_reader :times
  attr_reader :matrix

  #------------------------------------------------------------
  # Initializes the Trajectories class.  Takes two parameters
  # @param times
  # @param matrix
  #------------------------------------------------------------

  def initialize(times: 1, matrix: R.matrix(0))
    @times = times
    @matrix = matrix
  end

end

@traj_bis = TrajectoriesBis.new
```

Let's take a look at our new class:

```
puts @traj_bis.times
puts
puts @traj_bis.matrix
```

```
## 1
##
##      [,1]
## [1,]    0
```

Note that '@traj_bis.times' is the numeric 1, and what we actually want is a vector with [1] in it.

```
class TrajectoriesBis

  attr_reader :times
  attr_reader :matrix

  #------------------------------------------------------------
  # Initializes the Trajectories class.  Takes two parameters
  # @param times [R::Vector] should be an R vector.
  # @param matrix [R::Matrix] should be an R matrix.
  #------------------------------------------------------------

  # Use R.c to convert number 1 to a vector
  def initialize(times: R.c(1), matrix: R.matrix(0))
    @times = times
    @matrix = matrix
  end
```

```
end

@traj_bis = TrajectoriesBis.new
```

```
puts @traj_bis.times
puts
puts @traj_bis.matrix
```

```
## [1] 1
##
##      [,1]
## [1,]    0
```

8 The Empty Object

When a Trajectories is created with new, and no argument is given, all its instance variables will have the default nil value. Since Ruby has no type information, then there is only one type (or actually no type) of nil. To check if a variable is empty, we check it against the nil value.

9 To See an Object

Ruby has very strong meta-programming features, in particular, one can use introspection to see methods and instance variables from a given class. Method 'instance_variables' shows all the instance variables of an object:

```
puts @traj.instance_variables
```

The description of all meta-programming features of Ruby is well beyond the scope of this document, but it is a very frequent a powerful feature of Ruby, that makes programming in Ruby a different experience than programming in other languages.

10 Methods

Methods are a fundamental feature of object oriented programming. We will now extend our class Trajectories to add methods to it. In SS4, a method 'plot' is added to Trajectories. At this point, Renjin and Galaaz do not yet have plotting capabilities, so we will have to skip this method and go directly to the implementation of the 'print' method.

Bellow is the R code for method print:

```
> setMethod ("print","Trajectories",
+ function(x,...){
+ cat("*** Class Trajectories, method Print *** \\n")
+ cat("* Times ="); print (x@times)
+ cat("* Traj = \\n"); print (x@traj)
+ cat("******* End Print (trajectories) ******* \\n")
+ }
+ )
```

Now the same code for class Trajectories in Galaaz. In general methods are defined in a class together with all the class definition. We will first use this approach. Later, we will show how to 'reopen' a class to add new methods to it.

In this example, we are defining a method named 'print'. We have being using method 'puts' to output data. There is a Ruby method that is more flexible than puts and that we need to use to implement our function: 'print'. However, trying to use Ruby print inside the definition of Trajectories's print will not work, as Ruby will understand that as a recursive call to print. Ruby's print is defined inside the Kernel class, so, in order to call Ruby's print inside the definition of Trajectories's print we need to write 'Kernel.print'.

```
class Trajectories

  attr_reader :times
  attr_reader :matrix


  #------------------------------------------------------------
  # Initializes the Trajectories class.  Takes two parameters
  # @param times [R::Vector] should be an R vector.
  # @param matrix [R::Matrix] should be an R matrix.
  #------------------------------------------------------------

  def initialize(times: nil, matrix: nil)
    @times = times
    @matrix = matrix
  end

  #------------------------------------------------------------
  #
  #------------------------------------------------------------

  def print
    puts("*** Class Trajectories, method Print *** ")
    Kernel.print("times = ")
    puts @times
    puts("traj =")
    puts @matrix
    puts("******* End Print (trajectories) ******* ")
  end

end
```

```
@trajCochin.print
```

```
## *** Class Trajectories, method Print ***
## times = 1
## 3
## 4
## 5
## traj =
##          [,1] [,2] [,3] [,4]
```

```
## g2_v2114 15.0 15.1 15.2 15.2
## g2_v2115 16.0 15.9 16.0 16.4
## g2_v2116 15.2   NA 15.3 15.3
## g2_v2117 15.7 15.6 15.8 16.0
## ******* End Print (trajectories) *******
```

For Cochin, the result is correct. For Saint-Anne, print will display too much information. So we need a second method.

Show is the default R method used to show an object when its name is written in the console. We thus define 'show' by taking into account the size of the object: if there are too many trajectories, 'show' posts only part of them.

Here is the R code for method 'show':

```
> setMethod("show","Trajectories",
+ function(object){
+ cat("*** Class Trajectories, method Show *** \\n")
+ cat("* Times ="); print(object@times)
+ nrowShow <- min(10,nrow(object@traj))
+ ncolShow <- min(10,ncol(object@traj))
+ cat("* Traj (limited to a matrix 10x10) = \\n")
+ print(formatC(object@traj[1:nrowShow,1:ncolShow]),quote=FALSE)
+ cat("******* End Show (trajectories) ******* \\n")
+ }
+ )
```

Now, let's write it with Galaaz. This time though, we will not rewrite the whole Trajectories class, but just reopen it to add this specific method. The next example has many interesting features of Galaaz, some we have already seen, others will be described now:

- As we have already seen, to call an R function one uses the R. notation. There is however another way: when the first argument to the R function is an R object such as a matrix, a list, a vector, etc. we can use '.' notation to call the function. This makes the function look like a method of the object. For instance, R.nrow(@matrix), can be called by doing @matrix.nrow;
- In R, every number is converted to a vector and this can be done with method R.i. Converting a vector with only one number back to a number can be done with method '.gz'. So if @num is an R vector that holds a number, then @num.gz is a number that can be used normally with Ruby methods;
- R functions and Ruby methods can be used freely in Galaaz. We show bellow two different ways of getting the minimum of a number, either by calling R.min or by getting the minimum of an array, with the min method;
- Galaaz allows for method 'chaining'. Method chaining, also known as named parameter idiom, is a common syntax for invoking multiple method calls in object-oriented programming languages. Each method returns an object, allowing the calls to be chained together in a single statement without requiring variables to store the intermediate results. For instance @matrix.nrow.gz, which returns the number of rows of the matrix as a number;

- Ranges in Ruby are represented by (x..y), where x is the beginning of the range and y its end. An R matrix can be indexed by range, object@traj[1:nrowShow,1:ncolShow], the same result is obtained in Galaaz by indexing @matrix [(1..nrow_show), (1..ncol_show)]. Observe that this statement is then chained with the format function and with the pp method to print the matrix.

```
class Trajectories

  #------------------------------------------------------------
  #
  #------------------------------------------------------------

  def show
    puts("*** Class Trajectories, method Show *** ")
    Kernel.print("times = ")
    puts @times
    nrow_show = [10, @matrix.nrow << 0].min
    ncol_show = R.min(10, @matrix.ncol) << 0
    puts("* Traj (limited to a matrix 10x10) = ")
    puts @matrix[(1..nrow_show), (1..ncol_show)].format(digits: 2, nsmall: 2)
    puts("******* End Show (trajectories) ******* ")
  end

end
```

```
@trajStAnne.show
```

```
## parse error
```

```
## /home/rbotafogo/desenv_linux/galaaz/lib/new_bridge/session_client.rb:268:in 'eval_r'
## /home/rbotafogo/desenv_linux/galaaz/lib/R_interface/new_bridge_adapter.rb:231:in 'eval_r
## /home/rbotafogo/desenv_linux/galaaz/lib/R_interface/rsupport.rb:359:in 'exec_function'
## /home/rbotafogo/desenv_linux/galaaz/lib/R_interface/rsupport.rb:618:in 'process_missing_
## /home/rbotafogo/desenv_linux/galaaz/lib/R_interface/rsupport.rb:468:in 'process_missing'
## /home/rbotafogo/desenv_linux/galaaz/lib/R_interface/robject.rb:404:in 'method_missing'
## /home/rbotafogo/desenv_linux/galaaz/lib/util/exec_ruby.rb:180:in 'show'
## /home/rbotafogo/desenv_linux/galaaz/lib/util/exec_ruby.rb:170:in 'exec_ruby'
## org/jruby/RubyKernel.java:1268:in 'eval'
## /home/rbotafogo/desenv_linux/galaaz/lib/util/exec_ruby.rb:169:in 'exec_ruby'
## /home/rbotafogo/desenv_linux/galaaz/lib/gknit/knitr_engine.rb:777:in 'block in initialize
## org/jruby/RubyBasicObject.java:2695:in 'instance_eval'
## org/jruby/RubyBasicObject.java:2723:in 'instance_eval'
## /home/rbotafogo/desenv_linux/galaaz/lib/gknit/knitr_engine.rb:748:in 'block in initialize
## /home/rbotafogo/desenv_linux/galaaz/lib/R_interface/new_bridge_adapter.rb:358:in 'block
## /home/rbotafogo/desenv_linux/galaaz/lib/new_bridge/session_client.rb:413:in 'block in har
```

Our show method has the same problem as SS4, i.e., if an empty trajectories object is created and we try to 'show' it, it will generate an error. Let's see it:

```
@empty_traj = Trajectories.new
```

```
@empty_traj.show
```

```
## undefined method 'nrow' for nil
```